To make the N-step update work off-policy using importance sampling, we can use the likelihood ratio between the policy we want to learn (π) and the behavior policy (π') from which the data was drawn.

The typical N-step TD update for a Q-function can be represented as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\sum_{i=0}^{N-1} \gamma^i r_{t+i} + \gamma^N Q(s_{t+N}, a_{t+N}) - Q(s_t, a_t) \right)$$

(Where α is the learning rate, γ is the discount factor, and r are rewards.)

To adjust this for off-policy learning using importance sampling, we use the likelihood ratio:

$$\rho_i = \frac{\pi(a_{t+i}|s_{t+i})}{\pi'(a_{t+i}|s_{t+i})}$$

Our update becomes:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\sum_{i=0}^{N-1} \prod_{j=0}^{i} \rho_j \gamma^i r_{t+i} + \prod_{j=0}^{N-1} \rho_j \gamma^N Q(s_{t+N}, a_{t+N}) - Q(s_t, a_t) \right)$$

Now, for your questions:

Do we need to change (2) when $N = 1$?

When $N = 1$, the update becomes a one-step TD update. In this case, only the immediate reward and the Q-value of the next state-action pair are considered. If you're trying to correct for the off-policy nature of the data, even with $N = 1$, you'd need to adjust the update with the likelihood ratio for that single step:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\rho_0 r_t + \rho_0 \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right)$$

What about as $N \to \infty$?

As $N \to \infty$, you're looking at an infinite horizon, and the N-step return becomes the total return from state s_t onward. In practice, it's computationally infeasible to have infinite rollouts. However, conceptually, the importance sampling corrections would still be applied at each step in the sum. As the horizon grows, the product of the likelihood ratios could lead to high variance in the updates, which is a known issue with importance sampling in off-policy learning. It's one of the reasons why techniques like truncated importance sampling or other variance reduction techniques might be employed in such settings.

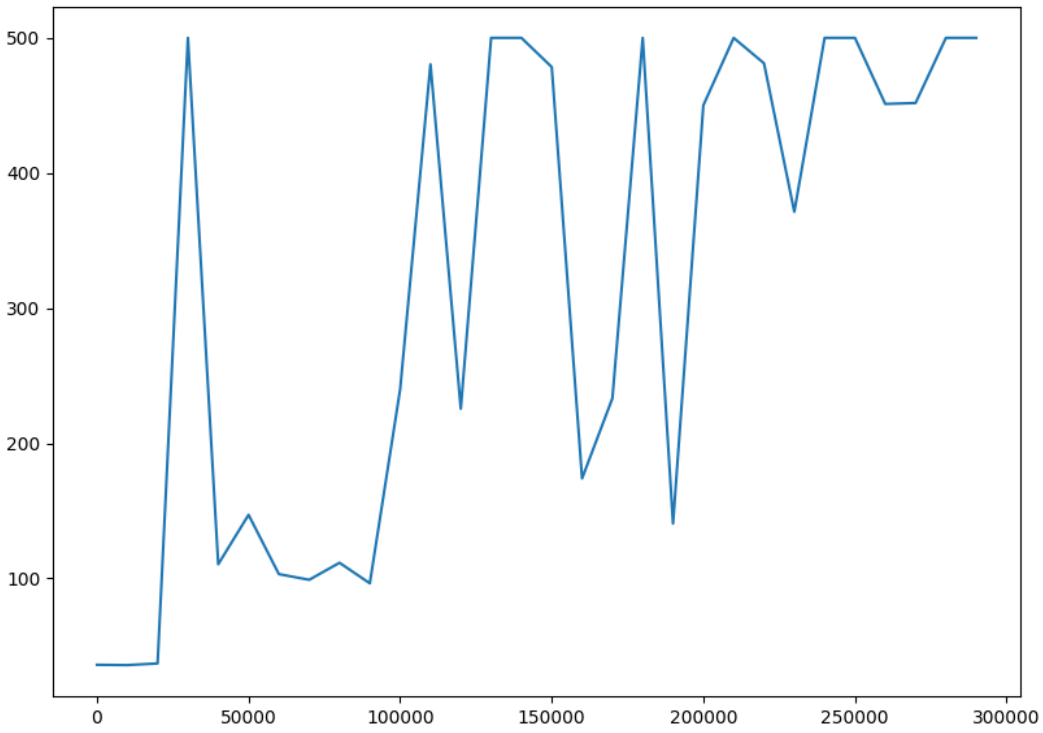
HW3 Writeup

Bo-Yen Chang

2.4

2.4.1 - Q1

- Submit your logs of CartPole-v1, and a plot with environment steps on the x-axis and eval return on the y-axis.

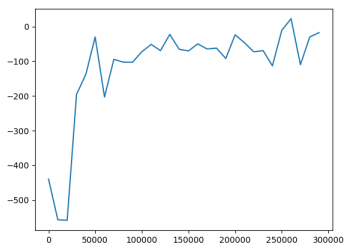


CartPole-v1

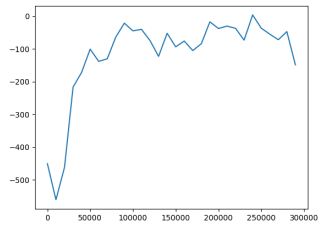
2.4.2 - Q2

Run DQN with three different seeds on LunarLander-v2:

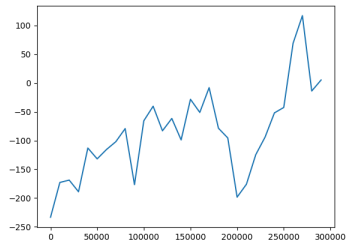
✍️ python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 1
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 2
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 3



Seed1



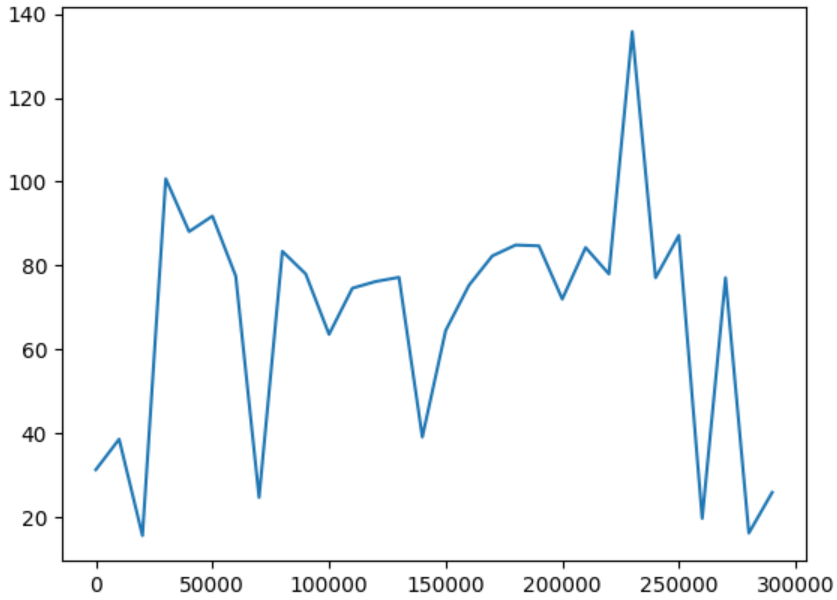
Seed2



Seed3

2.4.3 - Q3

Run DQN on CartPole-v1, but change the learning rate to 0.05 (you can change this in the YAML
config file). What happens to (a) the predicted Q-values, and (b) the critic error? Can you relate this to any topics from class or the analysis section of this homework?



Q. Run DQN on CartPole-v1, but change the learning rate to 0.05 (you can change this in the YAML config file). What happens to (a) the predicted Q-values, and (b) the critic error?

✍🏻 This is a graphical representation of the results for the Deep Q-Network (DQN) on the CartPole-v1 task, where the learning rate is set at 0.05. From the graph, we can observe the following outcomes:

1. **Predicted Q-values**: From the chart, the Q-values undergo rapid increases and decreases at certain intervals. When the learning rate is set at 0.05, the Q-values might experience larger fluctuations, as a higher learning rate can lead to instability during the optimization process of the Q-values. At certain times, there might be higher Q-value predictions, implying the network believes the current policy is optimal; however, it might quickly decrease afterwards, possibly due to worse outcomes discovered during exploration.
2. **Critic's Error**: While it's not directly observable from the provided graph, the volatility in Q-values can indirectly suggest that the critic's error might also increase. With a higher learning rate, the predicted Q-values might swiftly deviate from the true Q-values, leading to an increased critic's error.

Furthermore, from classroom knowledge and analysis in assignments, we know that the learning rate is a crucial hyper-parameter. It determines how much of the new estimates we should consider during each update. An overly high learning rate can lead to unstable learning and higher critic errors because it might cause Q-value estimates to quickly diverge from their true values. Conversely, a too low learning rate can make the learning process overly slow and difficult to adapt to new environmental changes. Therefore, setting the learning rate correctly is key to achieving good learning performance.

In conclusion, an appropriate learning rate is vital. It ensures that the DQN can balance exploration and exploitation to achieve satisfactory learning results. From the provided graph, it's evident that using a 0.05 learning rate might result in instability in Q-values and the critic's error. Further finetuning may be required to find a more suitable learning rate.

Q. Can you relate this to any topics from class or the analysis section of this homework?

Relating to Class Topics and Analysis:

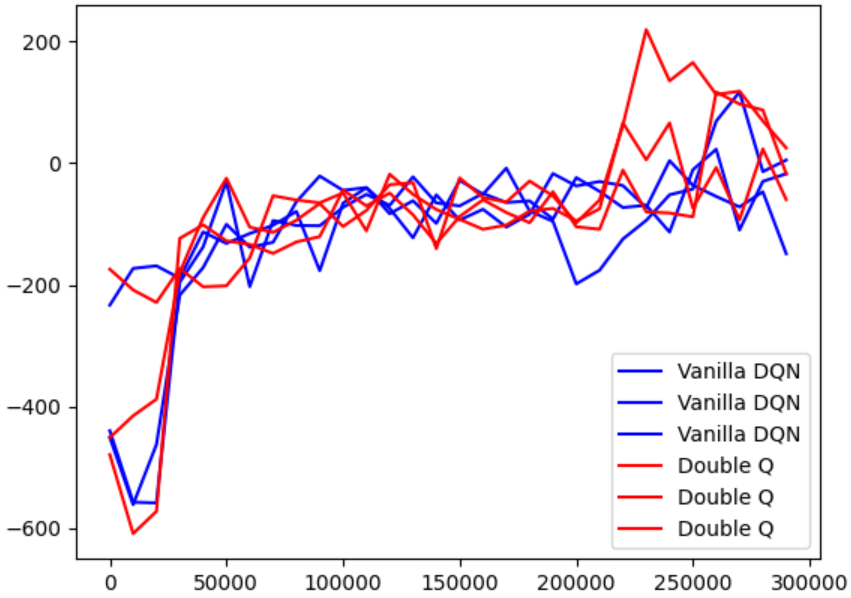
- **Stability vs. Speed of Convergence**: This trade-off is a common theme in reinforcement learning. A higher learning rate can speed up learning but may compromise stability. This is akin to concepts like "bias-variance tradeoff" in machine learning.
- **Learning Rate Schedules**: Sometimes, starting with a larger learning rate and reducing it over time can combine the best of both worlds—faster initial learning and stable convergence later on.
- **Exploration vs. Exploitation**: A highly fluctuating Q-value might lead to different action choices, increasing exploration. However, this is not a systematic way to handle exploration and is more of a side effect.

In summary, adjusting the learning rate in DQN will have a direct impact on the dynamics of learning, affecting both the predicted Q-values and the critic error. When analyzing or tuning hyperparameters like the learning rate, it's essential to consider the balance between stability and speed of convergence and how these choices might impact overall performance and learning dynamics.

2.5

2.5.1 - Q1

Q. Run three more seeds of the lunar lander problem. Plot returns from these three seeds in red, and the “vanilla” DQN results in blue, on the same set of axes.



Compare the two, and describe in your own words what might cause this difference.

1. **Convergence Speed:** The Double Q-learning lines (in red) seem to reach higher returns faster than the Vanilla DQN lines (in blue). This suggests that Double Q-learning might be learning a more effective policy quicker than Vanilla DQN.
2. **Stability:** The Double Q-learning curves appear more stable, especially towards the later parts of training. In contrast, the Vanilla DQN curves have more pronounced fluctuations. This can imply that Double Q-learning results in a more stable learning process, potentially due to its mechanism of mitigating overestimation bias.
3. **Final Performance:** By the end of training, both the Double Q-learning and Vanilla DQN seem to achieve similar high returns, though the former seems to do so more consistently across its different seeds.
4. **Early Divergence:** Early in the training, there's a significant divergence in returns between the different seeds of both algorithms, which gradually decreases as training progresses. This could be due to the initial randomness and exploration in the policies, which gets fine-tuned as training proceeds.

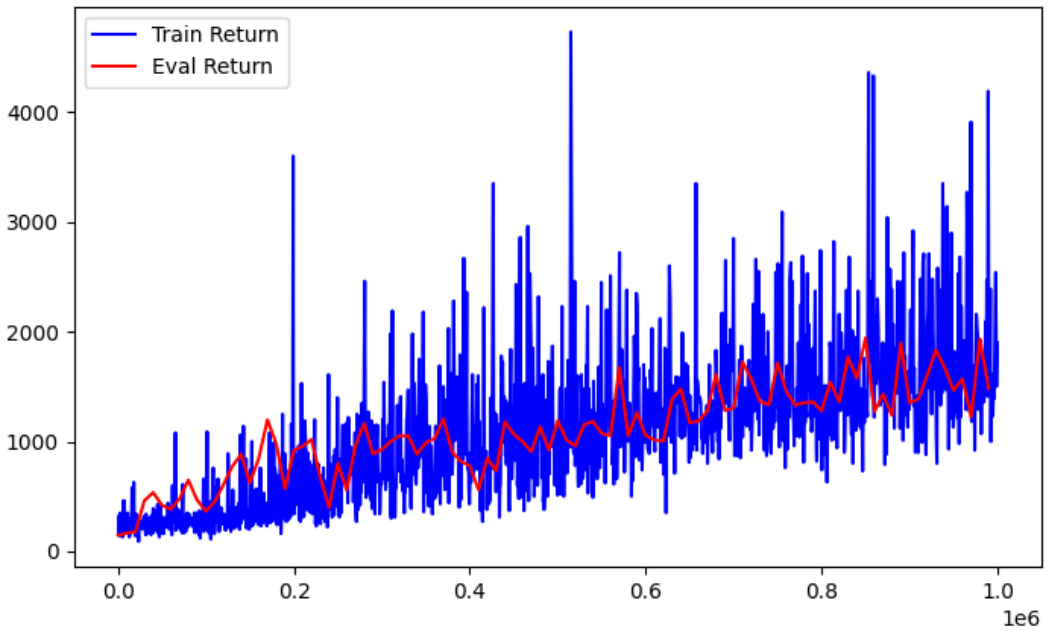
Reason for the observed difference:

Double Q-learning was proposed to address the overestimation bias of Q-values in standard DQN. In Vanilla DQN, the same network estimates both the current Q-values and the target Q-values, leading to potential overestimation. Overestimation can cause instability in training and suboptimal policies. Double Q-learning, on the other hand, uses two networks: one to select the action and another to evaluate that action's Q-value. This decoupling reduces the overestimation bias.

In simpler terms, Double Q-learning provides a "second opinion" on the value of an action, leading to more conservative and often more accurate Q-value estimates. This is likely why you observe quicker and more stable convergence in the Double Q-learning curves compared to Vanilla DQN.

2.5.3 - Q2, Q3 - lunarlander_doubleq

Q. **Plot the average training return (train_return) and eval return (eval_return) on the same axes. You may notice that they look very different early in training! Explain the difference.**



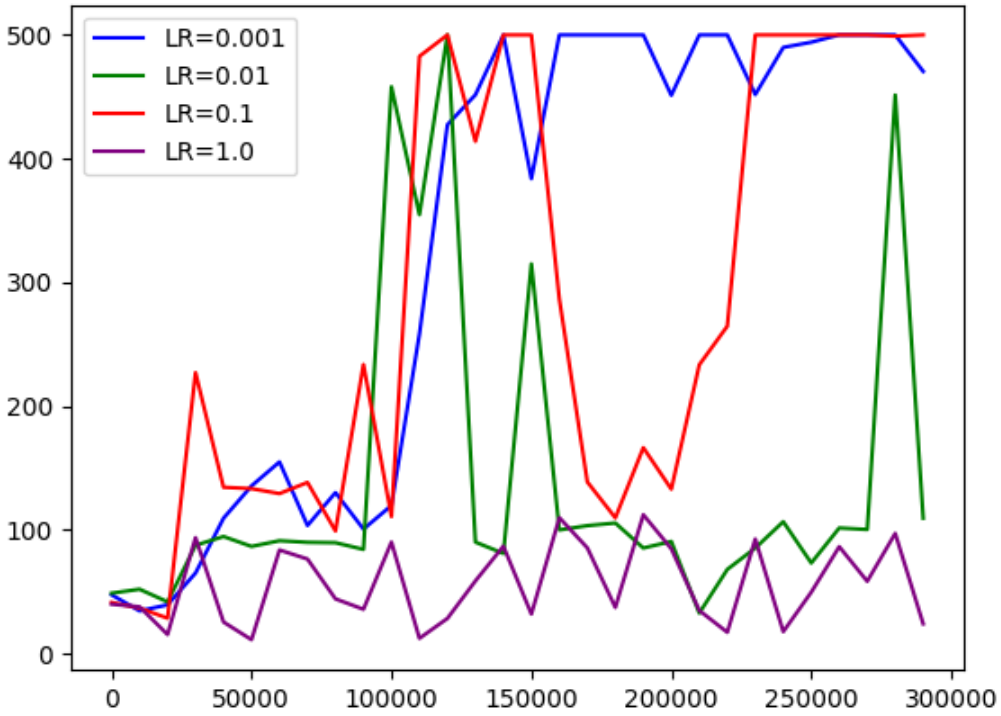
✍️ Difference between `train_return` and `eval_return` in the early stages of training: This is often due to the model optimizing on the training data it sees during the training period, while the evaluation return tests its performance on unseen data. Hence, the `train_return` might be higher as the model has optimized for that data. On the other hand, the `eval_return` might be lower as the model may not perform as well on unfamiliar data. This also reflects the situation of overfitting, where the model performs very well on the training data but not as well on the test data.

2.6

2.6.1 - Seed1, 2, 3 Experimenting with Hyperparameters

✍️ -lr 0.001, -lr 0.005, -lr 0.01, -lr 0.05

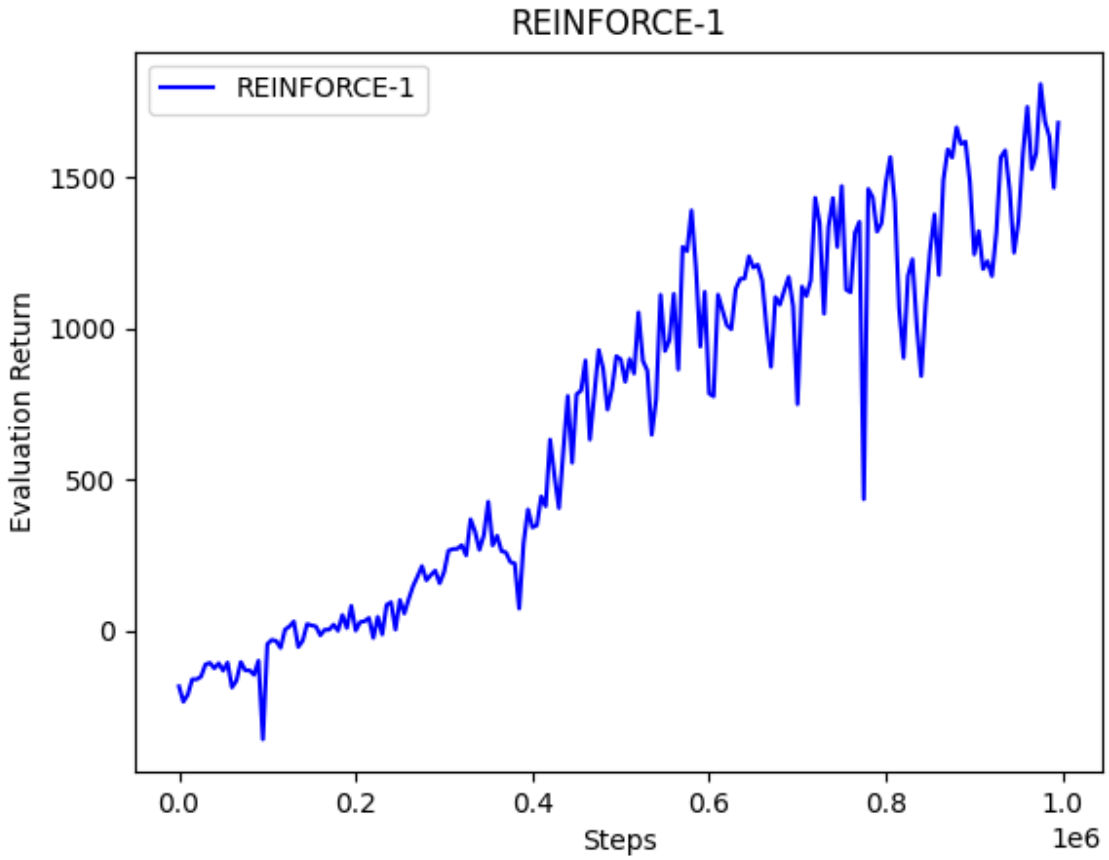
The **learning rate** is a pivotal hyper-parameter in deep learning, determining the magnitude of weight updates during each iteration. A well-chosen learning rate ensures optimal model performance and convergence speed. If set too high, the model might skip the best solution, leading to unstable training. In extreme cases, an excessive learning rate can deteriorate model performance as it might "jump" across the loss curve instead of smoothly approaching the optimal solution. Conversely, a learning rate set too low might cause the model to learn at a sluggish pace, necessitating more iterations to reach the best solution. In certain scenarios, an overly low learning rate can trap the model in a local minimum rather than a global one, preventing it from achieving peak performance.



3.1

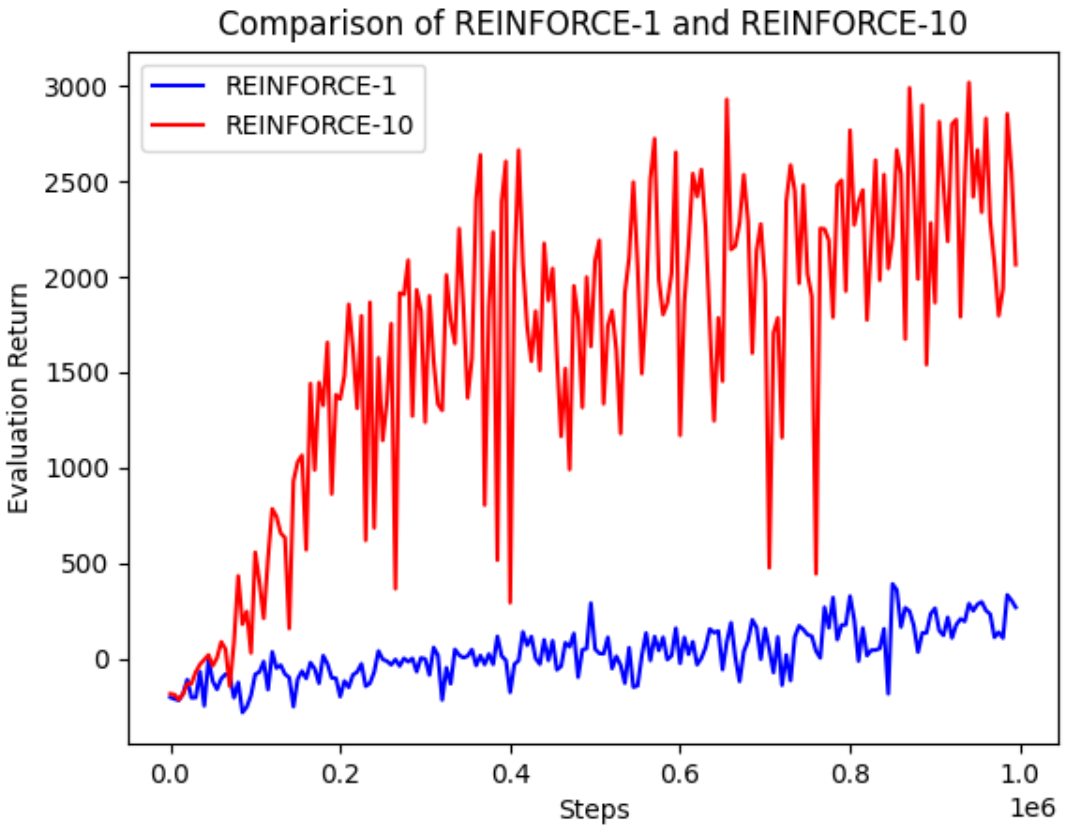
3.1.3 -Q1 REINFORCE

Q. Train an agent on HalfCheetah-v4 using the provided config halfcheetah_reinforce1.yaml). Note that this configuration uses only one sampled action per training example.



3.1.3 - Q2 REINFORCE

Q. Train another agent with halfcheetah_reinforce_10.yaml. This configuration takes many samples from the actor for computing the REINFORCE gradient (we'll call this REINFORCE-10, and the singlesample version REINFORCE-1). Plot the results (evaluation return over time) on the same axes as the single-sample REINFORCE. Compare and explain your results.



Compare and explain your results.

From the graph, we can make the following observations:

1. **Performance Difference**: REINFORCE-10 consistently outperforms REINFORCE-1. The evaluation return for REINFORCE-10 is much higher compared to REINFORCE-1 throughout the entire range of steps.
2. **Stability and Variability**: REINFORCE-10 shows greater variability in its performance, as indicated by the fluctuations in its evaluation return. On the other hand, REINFORCE-1 has a steadier curve, albeit with a much lower evaluation return.
3. **Convergence Rate**: Neither of the algorithms seems to have reached a plateau or converged to a stable value within the given number of steps. However, the gap in performance between them remains relatively consistent.

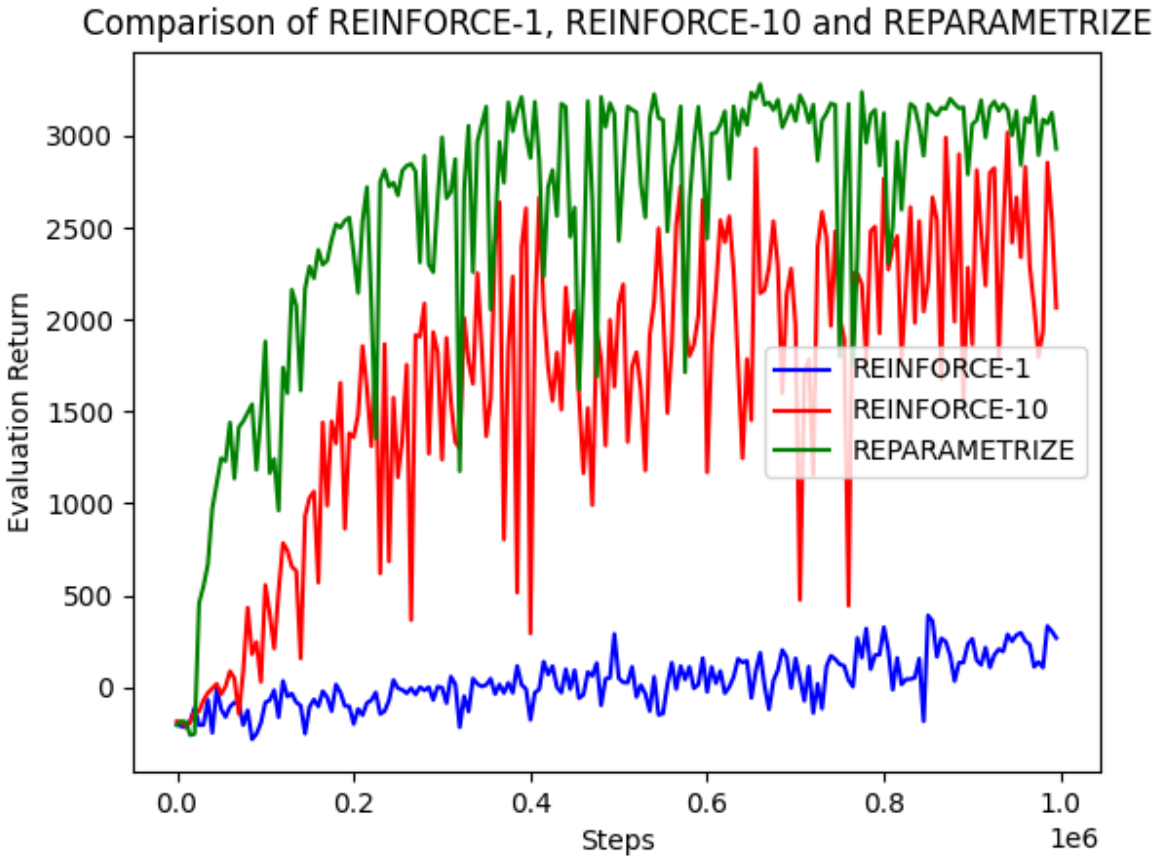
Explanation:

- The better performance of REINFORCE-10 might be attributed to its method of taking multiple samples from the actor for computing the REINFORCE gradient. This multiple sampling can provide a more accurate estimate of the gradient, leading to better policy updates.
- The increased variability in the performance of REINFORCE-10 may be a result of the higher variance introduced by multiple sampling. While it gives a better gradient estimate on average, individual estimates might vary significantly, leading to fluctuations in performance.
- The steadier performance curve of REINFORCE-1 could be due to the single-sample gradient estimation, which might have lower variance but could be biased or less accurate overall.

In conclusion, while REINFORCE-10 offers superior performance, it comes at the cost of higher variability. This suggests that while taking multiple samples can enhance the effectiveness of the REINFORCE gradient, it's essential to consider strategies to reduce the variance for more stable learning.

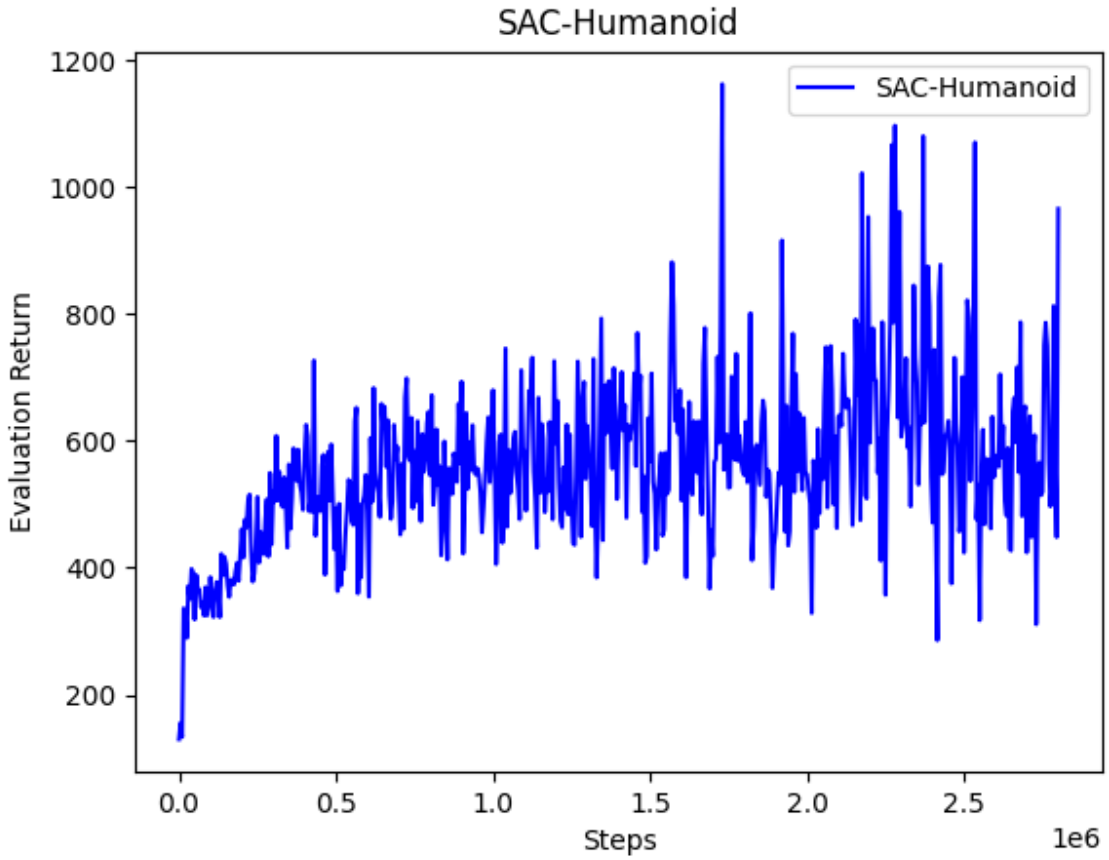
3.1.4 - Q1 REPARAMETRIZE

Q1. Train (once again) on HalfCheetah-v4 with halfcheetah_reparametrize.yaml. Plot results for all three gradient estimators (REINFORCE-1, REINFORCE-10 samples, and REPARAMETRIZE) on the same set of axes, with number of environment steps on the x-axis and evaluation return on the y-axis.



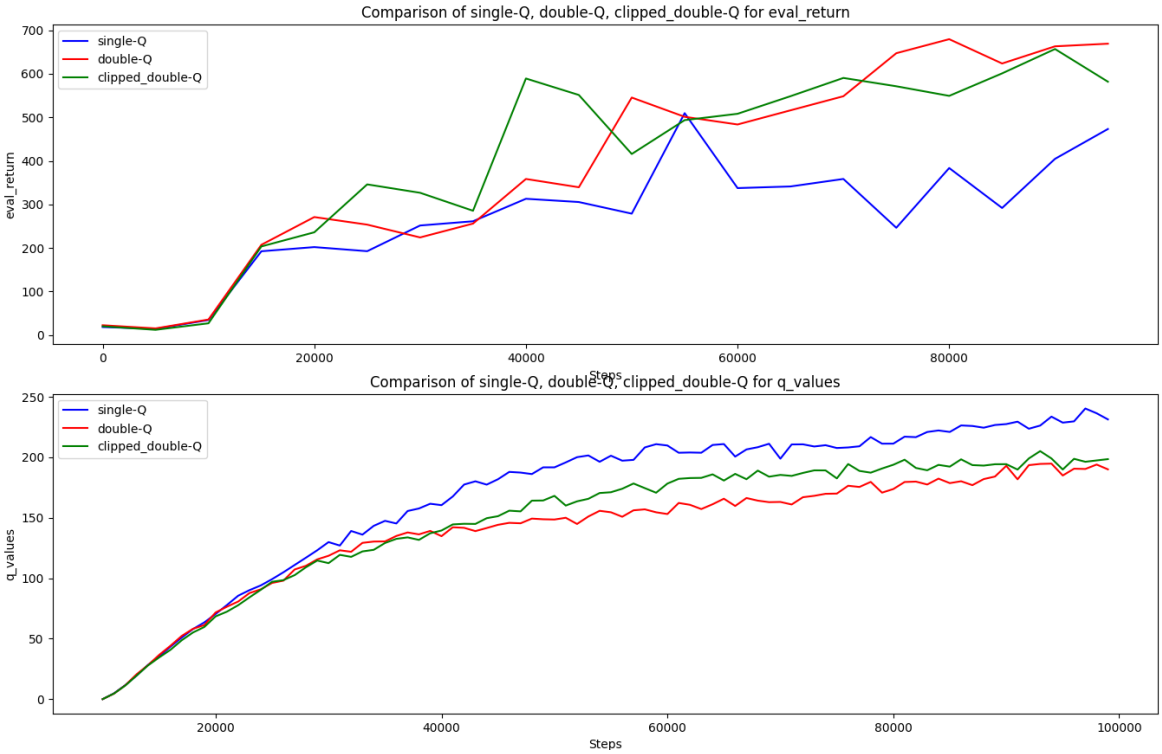
3.1.4 - Q2 REPARAMETRIZE

Q2. Train an agent for the Humanoid-v4 environment with humanoid_sac.yaml and plot results.



3.1.5 - Q1

Q1. Run single-Q, double-Q, and clipped double-Q on Hopper-v4 using the corresponding configuration files. Which one works best? Plot the logged eval_return from each of them as well as q_values.



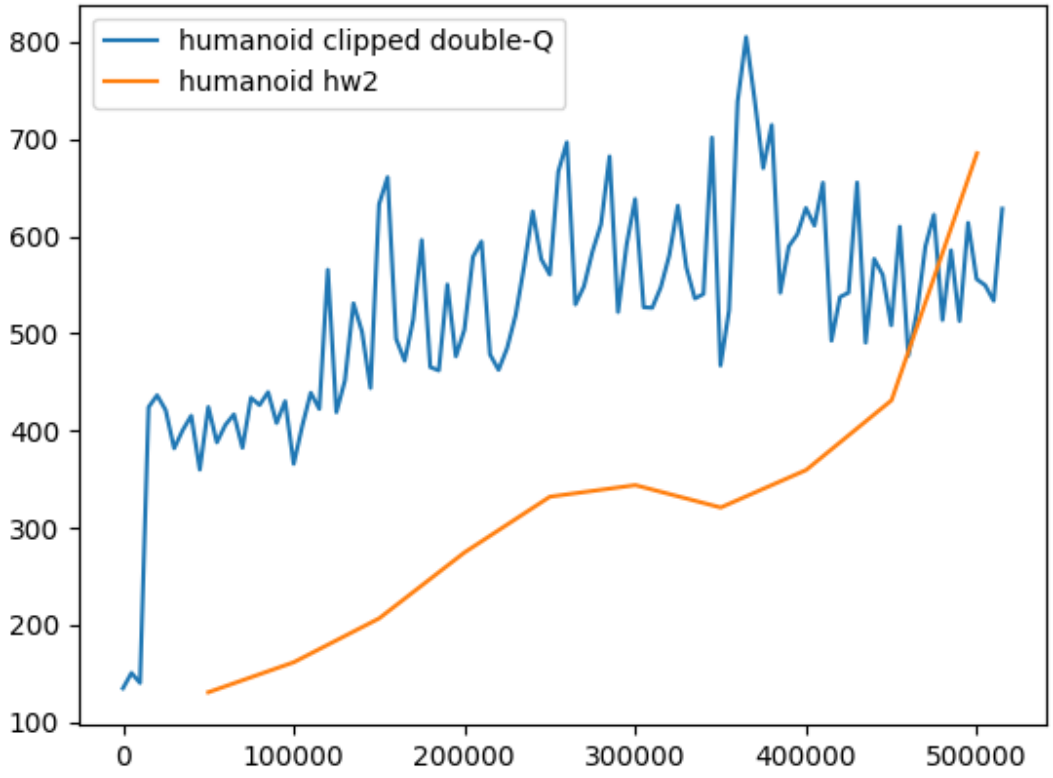
Discuss how these results relate to overestimation bias.

- **Best Performance**: Clipped double-Q has the highest evaluation return, suggesting that it performs best on Hopper-v4 among the three methods.
- **Overestimation Bias**: Overestimation bias refers to the phenomenon where algorithms tend to overestimate the Q-values of certain actions, leading to suboptimal policies. The decline in Q-values for double-Q around 80,000 steps might indicate a correction for overestimated Q-values. This overestimation might also explain the performance fluctuations seen in the double-Q evaluation return.
- **Clipped Double-Q Advantage**: Clipped double-Q seems to provide a balance, likely avoiding extreme overestimations by clipping the Q-values. This results in more consistent performance and higher evaluation returns.

In conclusion, clipped double-Q works best on Hopper-v4 among the tested methods. The observed results highlight the importance of addressing overestimation bias, with clipped double-Q seemingly offering a remedy by stabilizing Q-value estimates and providing better overall performance.

3.1.5 - Q2

Q. Pick the best configuration (single-Q/double-Q/clipped double-Q, or REDQ if you implement it) and run it on Humanoid-v4 using humanoid.yaml (edit the config to use the best option). You can truncate it after 500K environment steps. If you got results from the humanoid environment in the last homework, plot them together with environment steps on the x-axis and evaluation return on the y-axis.



Q. How do the off-policy and on-policy algorithms compare in terms of sample efficiency?

From the graph, we can infer the following about the sample efficiency of the off-policy and on-policy algorithms:

1. **Initial Growth**: The "humanoid clipped double-Q" (represents an off-policy algorithm) starts with a rapid increase in performance in the initial stages. This

suggests that it is quickly learning from its experience buffer and effectively utilizing past data for its updates.

2. **Stability and Progression**: The "humanoid hw2" (representing an on-policy method) exhibits a more steady and gradual increase in performance over the sampled steps. It does not have the same early spike in performance as the off-policy algorithm but shows continuous improvement over time.
3. **Peak Performance**: By the end of the sampled steps, the "humanoid clipped double-Q" has achieved a notably higher performance compared to "humanoid hw2". This indicates better sample efficiency for the off-policy algorithm, as it reaches higher returns with the same number of samples.

In conclusion, in terms of sample efficiency, the off-policy algorithm "humanoid clipped double-Q" appears to be more sample-efficient as it achieves higher returns faster and sustains that performance throughout the observation period. The on-policy "humanoid hw2", on the other hand, is slower in terms of extracting performance gains from the samples it collects.